Курс: Функциональное программирование

Лекция 10. Трансформеры монад

Денис Николаевич Москвин

02.12.2011

Кафедра математических и информационных технологий
Санкт-Петербургского академического университета

План лекции

- Снова моноиды
- Монада `Error`
- Монада `Cont`
- Трансформеры монад

План лекции

- Снова моноиды
- Монада `Error`
- Монада `Cont`
- Трансформеры монад

Класс `Monoid`

Некоторые аппликативные функторы и монады являются, помимо всего прочего, моноидами (списки, `Maybe`).

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Например,

```
instance Monoid a => Monoid (Maybe a) where
  mempty = Nothing
  Nothing `mappend` m = m
  m `mappend` Nothing = m
  Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Класс Alternative

```
class Applicative f => Alternative f where
    empty :: f a
    (<|>) :: f a -> f a -> f a

    -- One or more.
    some :: f a -> f [a]
    some v = some_v where
      many_v = some_v <|> pure []
      some_v = (:) <$> v <*> many_v
    -- Zero or more.
    many :: f a -> f [a]
    many v = many_v  where
      many_v = some_v <|> pure []
      some_v = (:) <$> v <*> many_v

infixl 3 <|>
```

Класс `Alternative`

```
instance Alternative Maybe where
    empty  =  Nothing

    Nothing <|> p  =  p
    Just x  <|> _  =  Just x
```

Возвращается первая «непустая» из возможных альтернатив

```
*Fp10> Nothing <|> (Just 3) <|> (Just 5) <|> Nothing
Just 3
```

Парсеры регулярных выражений: regex-applicative

`RE s a` — тип регулярного выражения, распознающего символы типа `s` и возвращающего результат типа `a`

```
psym :: (s -> Bool) -> RE s s
match :: RE s a -> [s] -> Maybe a

*Fp10> match (sym 'a' <|> sym 'b') "a"
Just 'a'
*Fp10> match (many $ psym isAlpha) "abc"
Just "abc"
*Fp10> match (many $ psym isAlpha) "123"
Nothing
*Fp10> match (many $ psym isAlpha) ""
Just ""
*Fp10> match (some $ psym isAlpha) ""
Nothing
```

Класс MonadPlus

```
class Monad m => MonadPlus m where
   mzero :: m a
   mplus :: m a -> m a -> m a

instance MonadPlus [] where
   mzero = []
   mplus = (++)

instance MonadPlus Maybe where
   mzero = Nothing

   xs      `mplus` _   = xs
   Nothing `mplus` ys  = ys
```

Эти представители имеют функциональность, аналогичную `Alternative`.

Использование MonadPlus

```
guard          :: (MonadPlus m) => Bool -> m ()
guard True     =  return ()
guard False    =  mzero

pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)

*Fp10> take 5 pythags
[(3,4,5),(6,8,10),(5,12,13),(9,12,15),(8,15,17)]
```

Использование `MonadPlus` (2)

```
msum  :: MonadPlus m => [m a] -> m a
msum  =  foldr mplus mzero

mfilter :: MonadPlus m => (a -> Bool) -> m a -> m a
mfilter p ma = do
  a <- ma
  if p a then return a else mzero
```

План лекции

- Снова моноиды
- Монада `Error`
- Монада `Cont`
- Трансформеры монад

Монада `Error`

Вычисление, которое может вызвать исключение.

```
class Error a where
    noMsg :: a
    strMsg :: String -> a

class (Monad m) => MonadError e m | m -> e where
    throwError :: e -> m a
    catchError :: m a -> (e -> m a) -> m a
```

Монада `Error`

```
instance MonadError (Either e) where
    throwError = Left
    (Left e) `catchError` handler = handler e
    a        `catchError` _       = a
```

Использование

```
do { action1; action2; action3 } `catchError` handler
```

План лекции

- Снова моноиды
- Монада `Error`
- Монада `Cont`
- Трансформеры монад

Стиль, основанный на предаче продолжений (1)

При использовании CPS результат вычисления функции не возвращается, а передаётся другой функции, которая получается исходной в качестве параметра ("продолжения").

```
square :: Int -> (Int -> r) -> r
square x k = k $ x^2

add :: Int -> Int -> (Int -> r) -> r
add x y k = k $ x + y

*Fp10> square 6 show
"36"
*Fp10> add 3 4 print
7
```

Последний аргумент — функция, получающая управление.

Стиль, основанный на предаче продолжений (2)

Вычисления конструируются как последовательности вложенных суб-вычислений

```
sum_squares :: Int -> Int -> (Int -> r) -> r
sum_squares x y k =
  square x  $ \x2 ->
  square y  $ \y2 ->
  add x2 y2 $ \ss ->
  k ss

*Fp10> sum_squares 3 4 print
25
```

Удобнее оформить в виде монады!

Монада `Cont`

```
newtype Cont r a = ...

cont :: ((a -> r) -> r) -> Cont r a
runCont :: Cont r a -> (a -> r) -> r

instance Monad (Cont r) where
  return x  =  cont $ \k -> k x
  m >>= f   =  cont $ \k -> runCont m
                      \a -> runCont (f a) k
```

Наш пример в терминах `Cont`

```
add' :: Int -> Int -> Cont r Int
add' x y = return $ x + y

square' :: Int -> Cont r Int
square' x = return $ x^2

sum_squares' :: Int -> Int -> Cont r Int
sum_squares' x y = do
  x2 <- square' x
  y2 <- square' y
  ss <- add' x2 y2
  return ss

*Fp10> runCont (sum_squares' 3 4) print
25
```

Функция `callCC` (call-with-current-continuation)

```
callCC :: ((a -> Cont r b) -> Cont r a) -> Cont r a

callCC $ \k -> ...

k :: a -> Cont r b
```

Простейший пример: сравним

```
square' :: Int -> Cont r Int
square' x = return $ x^2

square'' :: Int -> Cont r Int
square'' x = callCC $ \k -> k $ x^2
```

Функция `callCC` (2)

`callCC` обеспечивает механизм «выхода продолжением», прерывающим вычисление с немедленным возвратом значения.

```
foo :: Cont r Int
foo = callCC $ \k -> do
    let x = 2
    k x
    return 5 -- никогда!

roots :: (Double,Double,Double) -> Cont r String
roots (a,b,c) =  callCC $ \k -> do
    let delta = b^2 - 4 * a * c
    when (delta < 0) $ k "No roots!"
    let sqrt_delta = sqrt delta
    let x1 = (-b - sqrt_delta) / (2 * a)
    let x2 = (-b + sqrt_delta) / (2 * a)
    return $ "Roots are " ++ show x1 ++ ", " ++ show x2
```

Функция `callCC` (3)

На самом деле `callCC` определена в классе

```
class (Monad m) => MonadCont m where
    callCC :: ((a -> m b) -> m a) -> m a
```

Экземпляр для `(Cont r)` примерно такой

```
instance MonadCont (Cont r) where
    callCC f = Cont $ \c -> runCont (f (\a -> Cont $ \_ -> c a)) c
```

План лекции

- Снова моноиды
- Монада `Error`
- Монада `Cont`
- Трансформеры монад

Трансформеры монад: знакомство

```
stInteger :: State Integer Integer
stInteger = do modify (+1)
               a <- get
               return a

stString :: State String String
stString = do modify (++"1")
              b <- get
              return b

*Fp10> evalState stInteger 0
1
*Fp10> evalState stString "0"
"01"
```

Что делать если хотим в одном монадическом вычислении работать с обоими состояниями?

Трансформеры монад: знакомство

Monad transformers are like onions.

```
stComb :: StateT Integer (StateT String Identity) (Integer, String)
stComb = do modify (+1)
            lift $ modify (++"1")
            a <- get
            b <- lift $ get
            return (a,b)

*Fp10> runIdentity $ evalStateT (evalStateT stComb 0) "0"
(1,"01")
```

В качестве основы помимо `Identity` используют также `IO` со специализированной `liftIO`.

Трансформеры монад

Трансформер монад - конструктор типа, который принимает монаду в качестве параметра и возвращает монаду как результат.

Требования:

1. Поскольку у монады кайнд `m : * -> *`, у трансформера должен быть кайнд `t: (* -> *) -> * -> *`

2. Для любой монады `m`, аппликация `t m` должна быть монадой, то есть её `return` и `(>>=)` должны удовлетворять законам монад.

3. Нужен `lift :: m a -> t m a`, «поднимающий» значение из трансформируемой монады в трансформированную.

Рецепт приготовления трансформера для `MyMonad`

1. У трансформера должен быть кайнд `t: (* -> *) -> * -> * -> *`

Определяем наш конкретный трансформер `MyMonadT` для монады `MyMonad`

```
newtype MyMonadT m a = MyMonadT { runMyMonadT :: m (MyMonad a) }
```

Рецепт приготовления трансформера для `MyMonad`

2. Для любой монады `m`, аппликация `t m` должна быть монадой

Делаем аппликацию нашего трансформера к монаде (`MyMonadT m`) представителем `Monad`

```
instance (Monad m) => Monad (MyMonadT m) where
  return x    = ...
  mx (>>=) k  = ...
```

Рецепт приготовления трансформера для `MyMonad`

3. Операция `lift :: m a -> t m a` из `class MonadTrans`

Делаем наш трансформер `MyMonadT` представителем `MonadTrans`:

```
class MonadTrans t where
 lift :: (Monad m) => m a -> t m a

instance MonadTrans MyMonadT where
  lift mx = ...
```

Трансформер для Maybe

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance (Monad m) => Monad (MaybeT m) where
    fail _ = MaybeT (return Nothing)
    return = lift . return
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)

instance MonadTrans MaybeT where
    lift = MaybeT . liftM Just
```

Таблица стандартных трансформеров

Монада	Трансформер	Исходный тип	Тип трансформера
`Error`	`ErrorT`	`Either e a`	`m (Either e a)`
`State`	`StateT`	`s -> (a,s)`	`s -> m (a,s)`
`Reader`	`ReaderT`	`r -> a`	`r -> m a`
`Writer`	`WriterT`	`(a,w)`	`m (a,w)`
`Cont`	`ContT`	`(a -> r) -> r`	`(a -> m r) -> m r`

Они определены в библиотеке mtl. Более того, первый столбец определён через второй:

```
type State s = StateT s Identity
type Cont r = ContT r Identity
...
```

Что во что вкладывать?

Если нам нужна функциональность `Error` и `State`, то есть наша монада должна быть представителем `MonadError` и `MonadState`.

Должны ли мы применять трансформер `StateT` к монаде `Error` или трансформер `ErrorT` к монаде `State`?

Решение зависит от того, какой в точности семантики мы ожидаем от комбинированной монады.

Что во что вкладывать? (2)

Применение `StateT` к монаде `Error` даёт трансформирующую функцию типа `s -> Error e (a,s)`.

Применение `ErrorT` к монаде `State` даёт трансформирующую функцию типа `s -> (Error e a,s)`.

Порядок зависит от той роли, которую ошибка играет в вычислениях.

Если ошибка обозначает, что состояние не может быть вычислено, то нам следует применять `StateT` к `Error`.

Если ошибка обозначает, что значениене не может быть вычислено, но состояние при этом не «портится», то нам следует применять `ErrorT` к `State`.